

Table of Contents

<u>Best Practices</u>	1
<u>Streamlining File Transfers from Pleiades Compute Nodes to Lou</u>	1
<u>Increasing File Transfer Rates</u>	2
<u>Effective Use of Resources with PBS</u>	3
<u>Streamlining File Transfers from Pleiades Compute Nodes to Lou</u>	3
<u>Avoiding Job Failure from Overfilling /PBS/spool</u>	4
<u>Running Multiple Serial Jobs to Reduce Walltime</u>	5
<u>Checking the Time Remaining in a PBS Job from a Fortran Code</u>	8
<u>Memory Usage on Pleiades</u>	10
<u>Memory Usage Overview</u>	10
<u>Checking memory usage of a batch job using qps</u>	12
<u>Checking memory usage of a batch job using qtop.pl</u>	13
<u>Checking memory usage of a batch job using qsh.pl and "cat /proc/meminfo"</u>	14
<u>Checking memory usage of a batch job using gm.x</u>	15
<u>Checking if a Job was Killed by the OOM Killer</u>	17
<u>How to get more memory for your job</u>	19
<u>Lustre on Pleiades</u>	21
<u>Lustre Basics</u>	21
<u>Pleiades Lustre Filesystems</u>	24
<u>Lustre Best Practices</u>	27
<u>Lustre Filesystem Statistics in PBS Output File</u>	32

Best Practices

Streamlining File Transfers from Pleiades Compute Nodes to Lou

Some users prefer to streamline the storage of files (created during a job run) to Lou, within a PBS job. Since Pleiades compute nodes do not have network access to the outside world, all file transfers to Lou within a PBS job must go through the front-ends (pfe[1-12], bridge[1,2]) first.

Here is an example of what you can add to your PBS script to accomplish this:

1. Ssh to a front-end node (for example, bridge2) and create a directory on Lou where the files are to be copied.

```
ssh -q bridge2 "ssh -q lou mkdir -p $SAVDIR"
```

Here, \$SAVDIR is assumed to have been defined earlier in the PBS script. Note the use of *-q* for quiet-mode, and double quotes so that shell variables are expanded prior to the *ssh* command being issued.

2. Use scp via bridge[1,2] to transfer the files.

```
ssh -q bridge2 "scp -q $RUNDIR/* lou:$SAVDIR"
```

Here, \$RUNDIR is assumed to have been defined earlier in the PBS script.

Increasing File Transfer Rates

One challenge users face is moving large amounts of data efficiently to/from NAS across the network. Often, minor system, software, or network configuration changes can increase network performance an order of magnitude or more. This article describes some methods for increasing data transfer performance.

If you are experiencing slow transfer rates, try these quick tips:

- Transfer using the bridge nodes (bridge1, bridge2) instead of the Pleiades front-end systems (PFEs). The bridge nodes have much more memory, along with 10-Gigabit Ethernet interfaces to accommodate many large transfers. The PFEs often become oversubscribed and cause slowness.
- If using the scp command, make sure you are using OpenSSH version 5 or later. Older versions of SSH have a hard limit on transfer rates and are not designed for WAN transfers. You can check your version of SSH by running the command `ssh -V`.
- For large files that are a gigabyte or larger, we recommend using BBFTP. This application allows for transferring simultaneous streams of data and doesn't have the overhead of encrypting all the data (authentication is still encrypted).

Online Network Testing Tools

The [NAS PerfSONAR Service](#) provides a custom website that allows you to quickly self-diagnose your remote network connection issues, and reports the maximum bandwidth between sites, as well as any problems in the network path. Command-line tools are available if your system does not have a web browser.

Test results are also sent to our network experts, who will analyze traffic flows, identify problems, and work to resolve any bottlenecks that limit your network performance, whether the problem is at NAS or a remote site.

One-on-One Help

If you still require assistance in increasing your file transfer rates, please contact the NAS Control Room at support@nas.nasa.gov, and a network expert will work with you or your local administrator one-on-one to identify methods for increasing your rates.

To learn about other network-related support areas. see also, [End-to-End Networking Services](#).

Effective Use of Resources with PBS

Streamlining File Transfers from Pleiades Compute Nodes to Lou

Some users prefer to streamline the storage of files (created during a job run) to Lou, within a PBS job. Since Pleiades compute nodes do not have network access to the outside world, all file transfers to Lou within a PBS job must go through the front-ends (pfe[1-12], bridge[1,2]) first.

Here is an example of what you can add to your PBS script to accomplish this:

1. Ssh to a front-end node (for example, bridge2) and create a directory on Lou where the files are to be copied.

```
ssh -q bridge2 "ssh -q lou mkdir -p $SAVDIR"
```

Here, \$SAVDIR is assumed to have been defined earlier in the PBS script. Note the use of *-q* for quiet-mode, and double quotes so that shell variables are expanded prior to the *ssh* command being issued.

2. Use scp via bridge[1,2] to transfer the files.

```
ssh -q bridge2 "scp -q $RUNDIR/* lou:$SAVDIR"
```

Here, \$RUNDIR is assumed to have been defined earlier in the PBS script.

Avoiding Job Failure from Overfilling /PBS/spool

Before a PBS job is completed, its error and output files are kept in the /PBS/spool directory of the first node of your PBS job. The space under /PBS/spool is limited, however, and when it fills up, any job that tries to write to /PBS/spool may die. To prevent this, you should *not* write large amount of contents in the PBS output/error files.

If your executable normally produces a lot of output to the screen, you should redirect its output in your PBS script. For example:

```
#PBS ...
mpiexec a.out > output
```

To see the contents of your PBS output/error files before your job completes, follow the two steps below:

1. Find out the first node of your PBS job using "-W o=+rank0" for qstat:

```
%qstat -u your_username -W o=+rank0
JobID          User      Queue   Jobname    TSK  Nds      wallt  S      wallt    Eff  Rank0
-----
868819.pbsp11  zsmith   long    ABC         512   64  5d+00:00  R  3d+08:39 100%  r162i0n14
```

This shows that the first node is r162i0n14.

2. Log in to the first node and *cd* to /PBS/spool to find your PBS stderr/out file(s). You can view the content of these files using *vi* or *view*.

```
%ssh r162i0n14
%cd /PBS/spool
%ls -lrt
-rw----- 1 zsmith a0800 49224236 Aug  2 19:33 868819.pbsp11.nas.nasa.gov.OU
-rw----- 1 zsmith a0800 1234236 Aug  2 19:33 868819.pbsp11.nas.nasa.gov.ER
```

Running Multiple Serial Jobs to Reduce Walltime

DRAFT

This article is being reviewed for completeness and technical accuracy.

On Pleiades, running multiple serial jobs within a single batch job can be accomplished with following example PBS scripts. The maximum number of processes you can run on a single node will be limited to the core-count-per-node or the maximum number that will fit in a given node's memory, whichever is smaller.

processor type	cores/node	available memory/node
Harpertown	8	7.6 GB
Nehalem-EP	8	22.5 GB
Westmere-EP	12	22.5 GB

The examples below allow you to spawn serial jobs accross nodes using the `mpiexec` command. Note that a special version of `mpiexec` from the `mpi-mvapich2/1.4.1/intel` module is needed in order for this to work. This `mpiexec` keeps track of `$PBS_NODEFILE` and places each serial job onto the CPUs listed in `$PBS_NODEFILE` properly. The use of the arguments `"-comm none"` for this version of `mpiexec` is essential for serial codes or scripts. In addition, to launch multiple copies of the serial job at once, the use of the `mpiexec`-supplied `$MPIEXEC_RANK` environment variable is needed to distinguish different input/output files for each serial job. This is demonstrated with the use of a wrapper script `"wrapper.csh"` in which the input/output identifier (i.e., `${rank}`) is calculated from the sum of `$MPIEXEC_RANK` and an argument provided as input by the user.

Example 1:

This first example runs 64 copies of a serial job, assuming that 4 copies will fit in the available memory on one node and 16 nodes are used.

serial1.pbs:

```
#PBS -S /bin/csh
#PBS -j oe
#PBS -l select=16:ncpus=4
#PBS -l walltime=4:00:00

module load mpi-mvapich2/1.4.1/intel

cd $PBS_O_WORKDIR

mpiexec -comm none -np 64 wrapper.csh 0
```

wrapper.csh:

```
#!/bin/csh -f
@ rank = $1 + $MPIEXEC_RANK
./a.out < input_${rank}.dat > output_${rank}.out
```

This example assumes that input files are named input_0.dat, input_1.dat, ... and that they are all located in the directory where the PBS script is submitted from (i.e., \$PBS_O_WORKDIR). If the input files are in different directories, then wrapper.csh can be modified appropriately to cd into different directories as long as the directory names are differentiated by a single number that can be obtained from \$MPIEXEC_RANK (=0, 1, 2, 3, ...). In addition, be sure that wrapper.csh is executable by you and you have the current directory included in your path.

Example 2:

A second example provides the flexibility where the total number of serial jobs may not be the same as the total number of CPUs requested in a PBS job. Thus, the serial jobs are divided into a few batches and the batches are processed sequentially. Again, the wrapper script is used where multiple versions of the program "a.out" in a batch are run in parallel.

serial2.pbs:

```
#PBS -S /bin/csh
#PBS -j oe
#PBS -l select=10:ncpus=3
#PBS -l walltime=4:00:00

module load mpi-mvapich2/1.4.1/intel

cd $PBS_O_WORKDIR

# This will start up 30 serial jobs 3 per node at a time.
# There are 64 jobs to be run total, only 30 at a time.

# The number to run in total defaults here to 64 or the value
# of PROCESS_COUNT that is passed in via the qsub line like:
# qsub -v PROCESS_COUNT=48 serial2.pbs
#

# the total number to run at once is automatically determined
# at runtime by the number of cpus available.
# qsub -v PROCESS_COUNT=48 -l select=4:ncpus=3 serial2.pbs
# would make this 12 per pass not 30. no changes to script needed.

if ( $?PROCESS_COUNT ) then
    set total_runs=$PROCESS_COUNT
else
    set total_runs=64
endif

set batch_count=`wc -l < $PBS_NODEFILE`

set count=0
```

```
while ($count < $total_runs)
  @ rank_base = $count
  @ count += $batch_count
  @ remain = $total_runs - $count
  if ($remain < 0) then
    @ run_count = $total_runs % $batch_count
  else
    @ run_count = $batch_count
  endif
  mpiexec -comm none -np $run_count wrapper.csh $rank_base
end
```

Checking the Time Remaining in a PBS Job from a Fortran Code

DRAFT

This article is being reviewed for completeness and technical accuracy.

During job execution, sometimes it is useful to find out the amount of time remaining for your PBS job. This allows you to decide if you want to gracefully dump restart files and exit before PBS kills the job.

If you have an MPI code, you can call `MPI_WTIME` and see if the elapsed walltime has exceeded some threshold to decide if the code should go into the shutdown phase.

For example,

```
include "mpif.h"

real (kind=8) :: begin_time, end_time

begin_time=MPI_WTIME()
do work
end_time = MPI_WTIME()

if (end_time - begin_time > XXXXX) then
  go to shutdown
endif
```

In addition, the following library has been made available on Pleiades for the same purpose:

/u/scicon/tools/lib/pbs_time_left.a

To use this library in your Fortran code, you need to:

1. Modify your Fortran code to define an external subroutine and an integer*8 variable

```
external pbs_time_left
integer*8 seconds_left
```

2. Call the subroutine in the relevant code segment where you want the check to be performed

```
call pbs_time_left(seconds_left)
print*, "Seconds remaining in PBS job:", seconds_left
```

The return value from `pbs_time_left` is only accurate to within a minute or

3. Compile your modified code and link with the above library using, for example

```
LDFLAGS=/u/scicon/tools/lib/pbs_time_left.a
```

Memory Usage on Pleiades

Memory Usage Overview

Running jobs on cluster systems such as Pleiades requires more attention to the memory usage of a job than on shared memory systems. Below are a few factors that limit the amount of memory available to your running job:

- The total physical memory of a Pleiades compute node varies from 8 GB to 24 GB. A small amount of the physical memory is used by the system kernel. Through PBS, a job can access up to about 7.6 GB of an 8-GB node (Harpertown) and about 22.5 GB of a 24-GB node (Nehalem-EP and Westmere-EP).
- The PBS prologue tries to clean up the memory used by the previous job that ran on the nodes of your current running job. If there is a delay in flushing the previous job's data from memory to disks (for example, due to Lustre issues), the actual amount of free memory available to your job will be less.
- I/O uses buffer cache that also occupies memory. If your job does a large amount of I/O, the amount of memory left for your running processes will be less.

If your job uses more than 1 node, beware that the memory usage reported in the PBS output file is not the total memory usage for your job: rather, it is the *memory used in the first node* of your job. To help you get a more accurate picture of the memory usage of your job, we provide a few in-house tools, listed below.

1. **qtop.pl** invokes *top* on the compute nodes of a job, and provides a snapshot of the amount of used and free memory of the whole node and the amount used by each running process. For more information, read the article [Checking Memory Usage of a Batch Job Using qtop.pl](#).
2. **qps** invokes *ps* on the compute nodes of a job, and provides a snapshot of the %mem used by its running processes. For more information, read the article [Checking Memory Usage of a Batch Job Using qps](#).
3. **qsh.pl** can be used to invoke the command *cat /proc/meminfo* on the compute nodes to provide a snapshot of the total and free memory in each node. For more information, read the article [Checking Memory Usage of a Batch Job Using qsh.pl and "cat /proc/meminfo"](#).
4. **gm.x** and **gm_post.x** provide the memory high water mark for each process of your job when the job finishes. For more information, read the article [Checking Memory Usage of a Batch Job Using gm.x](#).

These tools are installed under the directory `/u/scicon/tools/bin`. It is a good idea to include this directory in your path by modifying your shell startup script so that you don't have to provide the complete path name when using these tools. For example:

```
set path = ( $path /u/scicon/tools/bin )
```

If your job runs out of memory and is killed by the kernel, this event was probably recorded in system log files. Instructions on how to check whether this is the case are provided in the article [Checking if a Job was Killed by the OOM Killer](#).

If your job needs more memory, read the article [How to Get More Memory for your Job](#) for possible approaches.

Checking memory usage of a batch job using qps

User Jeff West provided us with a Perl script called *qps* (available under /u/scicon/tools/bin) that securely connects (via ssh) into each node of a running job and gets process status (*ps*) information on each node.

Syntax:

```
pfel% qps jobid
```

Example:

```
pfel% qps 26130
```

```
*** Job 26130, User abc, Procs 1
NODE      TIME      %MEM %CPU STAT TASK
r1i0n14  10:17:13    2.8 99.9 RL  ./a.out
r1i0n14  10:17:12    2.9 99.9 RL  ./a.out
r1i0n14  10:17:18    2.9 99.9 RL  ./a.out
r1i0n14  10:16:34    2.9 99.8 RL  ./a.out
r1i0n14  10:17:11    2.9 99.9 RL  ./a.out
r1i0n14  10:17:13    2.9 99.9 RL  ./a.out
r1i0n14  10:17:12    2.9 99.9 RL  ./a.out
r1i0n14  10:17:15    2.9 99.9 RL  ./a.out
```

Note: The % memory usage by a process reported by this script is the percentage of memory in *the whole node*. This script currently works only when users specify `ncpus=8` in the PBS resource request.

If you want to use *qps* to monitor the memory used by a job that requested a number of CPUs other than 8, then make a copy of the *qps* script and change that single occurrence of '8' on line 95 to the appropriate number of CPUs requested on each node.

Checking memory usage pf a batch job using qtop.pl

DRAFT

This article is being reviewed for completeness and technical accuracy.

A Perl script called *qtop.pl* (available under `/u/scicon/tools/bin`) was provided by Bob Hood of the NAS staff. This script ssh's into the nodes of a PBS job and performs the command *top*. The output of *qtop.pl* provides memory usage for the whole node and for each process.

Syntax:

```
pfel% qtop.pl [-b] [-p n] [-P s] [-h n] [-H s] [-t s] [-N s] PBSjobid
-b      : (for running in background or batch) don't run 'resize' command
-p n    : show at most n processes per host
-P s    : show only procs in s, a comma-separated list of ranges
          e.g. -P 1,8-9
-h      : don't show the column header line
-H s    : show only header lines in s, comma-separated ranges
          e.g. -H 1-2,7
          e.g. -H 0 (don't show any lines)
-t s    : pass string s (must be one argument) to top command
-n s    : show output only from nodes in s, comma-separated ranges
          e.g. -n 0,2-3                (relative node #'s)
-N s    : show output only from nodes in s, a comma-separated list
          e.g. -N r1i1n14,r1i1n15 (absolute node #'s)
```

Example: to skip the header and list 8 procs per host

```
pfel% qtop.pl -H 0 -p 8 996093
all nodes in job 996093:  r184i2n12
r184i2n12  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    20027 zsmith    25   0 23.8g 148m 5320 R  101  0.6   5172:37 a.out
    20028 zsmith    25   0 23.8g 140m 5140 R  101  0.6   5173:35 a.out
    20029 zsmith    25   0 23.9g 286m 6640 R  101  1.2   5172:23 a.out
    20030 zsmith    25   0 23.9g 245m 5040 R  101  1.0   5171:18 a.out
    20031 zsmith    25   0 23.9g 265m 6040 R  101  1.1   5171:46 a.out
    20032 zsmith    25   0 23.9g 246m 5300 R  101  1.0   5171:00 a.out
    20033 zsmith    25   0 23.8g 158m 5476 R  101  0.7   5172:41 a.out
    20034 zsmith    25   0 23.8g 148m 5280 R  101  0.6   5173:02 a.out
```

Checking memory usage of a batch job using qsh.pl and "cat /proc/meminfo"

DRAFT

This article is being reviewed for completeness and technical accuracy.

A Perl script called *qsh.pl* (available under `/u/scicon/tools/bin`) was provided by NAS staff member Bob Hood. This script ssh's into all the nodes used by a PBS job and runs a command that you supply.

Syntax:

```
pfel% qsh.pl pbs_jobid command
```

One good use of this script is to check the amount of free memory in the nodes of your PBS job.

Example:

```
pfel% qsh.pl 30329 "cat /proc/meminfo"

running "cat /proc/meminfo" on:  r56i2n14 r56i2n15
r56i2n14 :
  MemTotal:      8079728 kB
  MemFree:       857936 kB
  Buffers:        0 kB
  Cached:       3775472 kB
...
r56i2n15 :
  MemTotal:      8079728 kB
  MemFree:       5840920 kB
  Buffers:        0 kB
  Cached:       784280 kB
...
```

Checking memory usage of a batch job using gm.x

DRAFT

This article is being reviewed for completeness and technical accuracy.

NAS staff member Henry Jin created a tool called *gm.x* (available under `/u/scicon/tools/bin`) that reports the memory usage at the end of a run from each process.

Add `/u/scicon/tools/bin` to your `$PATH` so that you can invoke *gm.x* without the full path.

Use the `-h` option to find out what types of memory usage can be reported:

```
pfel%gm.x -h
gm - version 1.0
usage: gm.x [-opts] a.out [args]
    -hwm      ; high water mark (VmHWM)
    -rss      ; resident memory size (VmRSS)
    -wrss     ; weighted memory size (WRSS)
    -v        ; verbose flag
Default is by environment variable GM_TYPE (def=WRSS)
```

Note that the `-rss` option reports the last snapshot of resident set size usage captured by the kernel. With the `-wrss` option, *gm.x* calls the system function *get_weighted_memory_size*. More information about this function can be found from the man page **man get_weighted_memory_size**.

gm.x can be used for either OpenMP or MPI applications (linked with either SGI's MPT, MVAPICH or Intel MPI libraries) and you do not have to recompile your application for it. A script called *gm_post.x* then takes the per process memory usage information and computes the total memory used and the average memory used per process.

To use *gm.x* for an MPI code, add *gm.x* after the `mpiexec` options. For example:

```
mpiexec -np 4 gm.x ./a.out
Memory usage for (r1i1n0,pid=9767): 1.458 MB (rank=0)
Memory usage for (r1i1n0,pid=9768): 1.413 MB (rank=1)
Memory usage for (r1i1n0,pid=9770): 1.413 MB (rank=3)
Memory usage for (r1i1n0,pid=9769): 1.417 MB (rank=2)
```

```
mpiexec -np 4 gm.x ./a.out | gm_post.x
Number of nodes      = 1
Number of processes  = 4
Processes per node   = 4
Total memory         = 5.701 MB

Memory per node      = 5.701 MB
Minimum node memory  = 5.701 MB
```

Maximum node memory = 5.701 MB

Memory per process = 1.425 MB

Minimum proc memory = 1.413 MB

Maximum proc memory = 1.458 MB

If you use dplace to pin process, add *gm.x* after dplace:

```
mpiexec -np NN dplace -s1 gm.x ./a.out
```

Checking if a Job was Killed by the OOM Killer

If a PBS job runs out of memory and is killed by the Out-Of-Memory (OOM) killer of the kernel, this event is likely (though not always) recorded in system log files. You can confirm this event by checking some of the messages recorded in system log files, and then increase your memory request in order to get your job running.

Follow the steps below to check whether your job has been killed by the OOM killer:

1. Find out when your job ran, what rack numbers were used by your job, and if the job exited with the `Exit_status=137` from the `tracejob` output of your job. For example:

```
pfe[1-12]% ssh pbspl1
pbspl1% tracejob -n 3 140001
```

where "3" indicates that you want to trace your job (PBS JOBID=140001), which ran within the past 3 days.

2. From the rack numbers (such as `r2`, `r3`, ...), you then `grep` messages that were recorded in the messages file stored in the leader node of those racks for your executable. For example, to look at messages for rack `r2`:

```
pfe[1-12]% grep abc.exe /net/r2lead/var/log/messages
Apr 21 00:32:50 r2i2n7 kernel: abc.exe invoked oom-killer:
gfp_mask=0x201d2, order=0, oomkilladj=-17
```

3. Often, the Out-Of-Memory message doesn't make it into the messages file, but will be recorded in a consoles file named by each individual node. For example, to look for `abc.exe` invoking the OOM killer on node `r2i2n7`:

```
pfe% grep abc.exe /net/r2lead/var/log/consoles/r2i2n7
abc.exe invoked oom-killer: gfp_mask=0x201d2, order=0, oomkilladj=0
```

Note that these messages do not have a timestamp associated with them, so you will need to use an editor to view the file and look for the hourly time markers bracketing when the job ran out of memory. An hourly time marker looks like this:

```
[-- MARK -- Thu Apr 21 00:00:00 2011]
```

It's also possible that a system process (such as, `pbs_mom` or `ntpd`) is listed as invoking the OOM killer, but it is nevertheless direct evidence that the node had run out of memory.

If you want to monitor the memory use of your job while it is running, you can use the tools listed in the article [Memory Usage Overview](#).

In addition, NAS provides a script called *pbs_oom_check*. This script does the steps above and parses the `/var/log/messages` on all the nodes associated with `pbs_jobid`, looking for an instance of OOM killer. The script is available under `/u/scicon/tools/bin` and works best when run on the host `pbspl1`.

How to get more memory for your job

DRAFT

This article is being reviewed for completeness and technical accuracy.

If your job was terminated because it needed more memory than what's available in the nodes that it ran on, consider the following:

- Among the Harpertown nodes, the 64 nodes in rack 32 have 16 GB per node instead of 8 GB per node. You can request running your job on rack 32 with the keyword **bigmem=true**. For example, change

```
#PBS -lselect=1:ncpus=8
```

to

```
#PBS -lselect=1:ncpus=8:bigmem=true
```

- Run your job on Nehalem-EP or Westmere nodes instead of Harpertown nodes. For example, change

```
#PBS -lselect=1:ncpus=8:model=har
```

to

```
#PBS -lselect=1:ncpus=8:model=neh
```

or

```
#PBS -lselect=1:ncpus=8:model=wes
```

- If all processes use about the same amount of memory and you can not fit 8 processes per node (for Harpertown or Nehalem-EP, or 12 processes per node for Westmere-EP), reduce the number of processes per node and request more nodes for your job. For example, change

```
#PBS -lselect=3:ncpus=8:mpiprocs=8:model=neh
```

to

```
#PBS -lselect=6:ncpus=4:mpiprocs=4:model=neh
```

- For a typical MPI job where rank 0 does the I/O and uses a lot of buffer cache, assign rank 0 to 1 node by itself. For example, change

```
#PBS -lselect=1:ncpus=8:mpiprocs=8:model=neh
```

to

```
#PBS
```

```
-lselect=1:ncpus=1:mpiprocs=1:model=neh+1:ncpus=7:mpiprocs=7:model=neh
```

Due to formatting issue, the above may appear as 2 lines. It should really be just 1 line.

- If you suspect that certain nodes that your job ran on had less total physical memory than normal, report it to NAS Help Desk. Those nodes can be offlined and taken care of by NAS staff. This prevents you and other users from using those nodes before they are fixed.
- For certain pre- or post-processing work that needs more than 22.5 GB of memory, run it on the bridge nodes (bridge[1,2]) interactively. Note that jobs running on the bridge nodes can not use more than 48 GB of memory. Also MPI applications that use SGI's MPT library can not run on the bridge nodes.
- For a multi-process or multi-thread job, if any of your processes/threads needs more than 22.5 GB, it won't run on Pleiades. Run it on a shared memory system such as Columbia.

Lustre on Pleiades

Lustre Basics

DRAFT

This article is being reviewed for completeness and technical accuracy.

A Lustre filesystem is a high-performance, shared filesystem (managed with the Lustre software) for Linux clusters. It is highly scalable and can support many thousands of client nodes, petabytes of storage and hundreds of gigabytes per second of I/O throughput.

Main Lustre components:

- Metadata Server (MDS)

1 or 2 per filesystem; service nodes that manage all metadata operations such as assigning and tracking the names and storage locations of directories and files on the OSTs.

- Metadata Target (MDT)

1 per filesystem; a storage device where the metadata (name, ownership, permissions and file type) are stored.

- Object Storage Server (OSS)

1 or multiple per filesystem; service nodes that run the Lustre software stack, provide the actual I/O service and network request handling for the OSTs, and coordinate file locking with the MDS. Each OSS can serve up to ~15 OSTs. The aggregate bandwidth of a Lustre filesystem can approach the sum of bandwidths provided by the OSSes.

- Object Storage Target (OST)

multiple per filesystem; storage devices where the data in user files are stored. Under Linux 2.6 (current OS on Pleiades), each OST can be up to 8TB in size. Under SLES 11, each OST can be up to 16 GB in size. The capacity of a Lustre filesystem is the sum of the sizes of all OSTs.

- Lustre Clients

commonly in the thousands per filesystem; compute nodes that mount the Lustre filesystem, and access/use data in the filesystem.

Striping

A user file can be divided into multiple chunks and stored across a subset of the OSTs. The chunks are distributed among the OSTs in a round-robin fashion to ensure load balancing.

Benefits of striping:

- allows one to have a file size larger than the size of an OST
- allows one or more clients to read/write different parts of the same file at the same time and provide higher I/O bandwidth to the file since the bandwidth is aggregated over the multiple OSTs

Drawbacks of striping:

- higher risk of file damage due to hardware malfunction
- increased overhead due to network operations and server contention

There are default stripe configurations for each Lustre filesystem. However, users can set the following stripe parameters for their own directories or files to get optimum I/O performance:

1. stripe_size

the size of the chunk in bytes; specify with k, m, or g to use units of KB, MB, or GB, respectively; the size must be an even multiple of 65,536 bytes; default is 4MB for all Pleiades Lustre filesystems; one can specify 0 to use the default size.

2. stripe_count

the number of OSTs to stripe across; default is 1 for most of Pleiades Lustre filesystems (/nobackupp[10-60]); one can specify 0 to use the default count; one can specify -1 to use all OSTs in the filesystem.

3. stripe_offset

The index of the OST where the first stripe is to be placed; default is -1 which results in random selection; using a non-default value is NOT recommended.

Use the command for setting the stripe parameters:

```
pfel% lfs setstripe -s stripe_size -c stripe_count -o stripe_offset  
dir|filename
```

For example, to create a directory called dir1 with a stripe_size of 4MB and a stripe_count of 8, do

```
pfel% mkdir dir1
```

```
pfel% lfs setstripe -s 4m -c 8 dir1
```

Also keep in mind that:

- When a file or directory is created, it will inherit the parent directory's stripe settings.
- The stripe settings of an *existing file* can not be changed. If you want to change the settings of a file, you can create a new file with the desired settings and copy the existing file to the newly created file.

Useful Commands for Lustre

- To list all the OSTs for the filesystem

```
pfel% lfs osts
```

- To list space usage per OST and MDT in human readable format for all Lustre filesystems or for a specific one, for example, /nobackupp10:

```
pfel% lfs df -h  
pfel% lfs df -h /nobackupp10
```

- To list inode usage for all filesystems or a specific one, for example, /nobackupp10:

```
pfel% df -i  
pfel% df -i /nobackupp10
```

- To create a new (empty) file or set directory default with specified stripe parameters

```
pfel% lfs setstripe -s stripe_size -c stripe_count -o  
stripe_offset dir|filename
```

- To list the striping information for a given file or directory

```
pfel% lfs getstripe dir|filename
```

- To display disk usage and limits on your /nobackup directory (for example, /nobackupp10):

```
pfel% lfs quota -u username /nobackupp10
```

or

```
pfel% lfs quota -u username /nobackup/username
```

To display usage on each OST, add the -v option:

```
pfel% lfs quota -v -u username /nobackup/username
```

Pleiades Lustre Filesystems

Pleiades has several Lustre filesystems (/nobackupp[10-60]) that provide a total of about 3 PB of storage and serve thousands of cores. These filesystems are managed under Lustre software version 1.8.2.

Lustre filesystem configurations are summarized at the end of this article.

Which /nobackup should I use?

Once you are granted an account on Pleiades, you will be assigned to use one of the Lustre filesystems. You can find out which Lustre filesystem you have been assigned to by doing the following:

```
pfel% ls -l /nobackup/your_username
lrwxrwxrwx 1 root root 19 Feb 23 2010 /nobackup/username -> /nobackupp30/username
```

In the above example, the user is assigned to /nobackupp30 and a symlink is created to point the user's default /nobackup to /nobackupp30.

TIP: Each Pleiades Lustre filesystem is shared among many users. To get good I/O performance for your applications and avoid impeding I/O operations of other users, read the articles: Lustre Basics and Lustre Best Practices.

Default Quota and Policy on /nobackup

Disk space and inodes quotas are enforced on the /nobackup filesystems. The default soft and hard limits for inodes are 75,000 and 100,000, respectively. Those for the disk space are 200GB and 400GB, respectively. To check your disk space and inodes usage and quota on your /nobackup, use the *lfs* command and type the following:

```
%lfs quota -u username /nobackup/username
Disk quotas for user username (uid xxxx):
    Filesystem  kbytes      quota   limit   grace   files   quota   limit   grace
/nobackup/username 1234  210000000 420000000    -     567   75000  100000    -
```

The NAS quota policy states that if you exceed the soft quota, an email will be sent to inform you of your current usage and how much of your grace period remains. It is expected that users will occasionally exceed their soft limit, as needed; however after 14 days, users who are still over their soft limit will have their batch queue access to Pleiades disabled.

If you anticipate having a long-term need for higher quota limits, please send a justification via email to support@nas.nasa.gov. This will be reviewed by the HECC Deputy Project Manager for approval.

For more information, see also, [Quota Policy on Disk Space and Files](#).

NOTE: If you reach the hard limit while your job is running, the job will die prematurely without providing useful messages in the PBS output/error files. A Lustre error with code -122 in the system log file indicates that you are over your quota.

In addition, when a Lustre filesystem is full, jobs writing to it will hang. A Lustre error with code -28 in the system log file indicates that the filesystem is full. The NAS Control Room staff normally will send out emails to the top users of a filesystem asking them to clean up their files.

Important: Backup Policy

As the names suggest, these filesystems are not backed up, so any files that are removed *cannot* be restored. Essential data should be stored on Lou1-3 or onto other more permanent storage.

Configurations

In the table below, /nobackupp[10-60] have been abbreviated as p[10-60].

Pleiades Lustre Configurations						
Filesystem	p10	p20	p30	p40	p50	p60
# of MDSES	1	1	1	1	1	1
# of MDTs	1	1	1	1	1	1
size of MDTs	1.1T	1.0T	1.2T	0.6T	0.6T	0.6T
# of usable inodes on MDTs	$\sim 235 \times 10^6$	$\sim 115 \times 10^6$	$\sim 110 \times 10^6$	$\sim 57 \times 10^6$	$\sim 113 \times 10^6$	$\sim 123 \times 10^6$
# of OSSes	8	8	8	8	8	8
# of OSTs	120	60	120	60	60	60
size/OST	7.2T	7.2T	3.5T	3.5T	7.2T	7.2T
Total Space	862T	431T	422T	213T	431T	431T
Default Stripe Size	4M	4M	4M	4M	4M	4M
Default Stripe Count	1	1	1	1	1	1

NOTE: The default stripe count and stripe size were changed on January 13, 2011. For directories created prior to this change, if you did not explicitly set the stripe count and/or stripe size, the default values (stripe count 4 and stripe size 1MB) were used. This means that files created prior to January 13, 2011 had those old default values. After this date, directories without an explicit setting of stripe count and/or stripe size adopted the new stripe count of 1 and stripe size of 4MB. However, the old files in that directory will retain their old default values. New files that you create in these directories will adopt the new

default values.

Lustre Best Practices

Lustre filesystems are shared among many users and many application processes, which causes contention for various Lustre resources. This article explains how Lustre I/O works, and provides best practices for improving application performance.

How does Lustre I/O work?

When a client (a compute node from your job) needs to create or access a file, the client queries the metadata server (MDS) and the metadata target (MDT) for the layout and location of the file's stripes. Once the file is opened and the client obtains the striping information, the MDS is no longer involved in the file I/O process. The client interacts directly with the object storage servers (OSSes) and object storage targets (OSTs) to perform the I/O operations such as locking, disk allocation, storage, and retrieval.

If multiple clients try to read and write the same part of a file at the same time, the Lustre distributed lock manager enforces coherency so that all clients see consistent results.

Jobs being run on Pleiades contend for shared resources in NAS's Lustre filesystem. The Lustre server can only handle about 15,000 remote procedure calls (RPCs, inter-process communications that allow the client to cause a procedure to be executed on the server) per second. Contention slows the performance of your applications and weakens the overall health of the Lustre filesystem. To reduce contention and improve performance, please apply the examples below to your compute jobs, while working in our high-end computing environment.

Best Practices

- **Avoid using *ls -l***

The *ls -l* command displays information such as ownership, permission and size of all files and directories. The information on ownership and permission metadata is stored on the MDTs. However, the file size metadata is only available from the OSTs. So, the *ls -l* command issues RPCs to the MDS/MDT and OSSes/OSTs for every file/directory to be listed. RPC requests to the OSSes/OSTs are very costly and can take a long time to complete for many files and directories.

- Use *ls* by itself if you just want to see if a file exists.
- Use *ls -l filename* if you want the long listing of a specific file.

- **Avoid having a large number of files in a single directory**

Opening a file keeps a lock on the parent directory. When many files in the same directory are to be opened, it creates contention. It is better to split a huge number of files (in the thousands or more) into multiple sub-directories to minimize contention.

- **Avoid accessing small files on Lustre filesystems**

Accessing small files on the Lustre filesystem is not efficient. If possible, keep them on an NFS-mounted filesystem (such as your home filesystem) or copy them from Lustre to /tmp on each node at the beginning of the job and access them from there.

- **Use a stripe count of 1 for directories with many small files**

If you have to keep small files on Lustre, be aware that *stat* operations are more efficient if each small file resides in one OST. Create a directory to keep small files, set the stripe count to 1 so that only one OST will be needed for each file. This is useful when you extract source and header files (which are usually very small files) from a tarfile.

```
pfel% mkdir dir_name
pfel% lfs setstripe -s 1m -c 1 dir_name
pfel% cd dir_name
pfel% tar -xf tarfile
```

If there are large files in the same directory tree, it may be better to allow them to stripe across more than one OST. You can create a new directory with a larger stripe count and copy the larger file to that directory. Note that moving files into that directory with the *mv* command will not change the stripe count of the files. Files must be created in or copied to a directory to inherit the stripe count properties of a directory.

```
pfel% mkdir dir_count_4
pfel% lfs setstripe -s 1m -c 4 dir_count_4
pfel% cp file_count_1 dir_count_4
```

If you have a directory with many small files (less than 100MB) and a few very large files (greater than 1GB), then it may be better to create a new subdirectory with a larger stripe count. Store just the large files and create symbolic links to the large files using the *symlink* command.

```
pfel% mkdir bigstripe
pfel% lfs setstripe -c 16 -s 4m bigstripe
pfel% ln -s bigstripe/large_file large_file
```

- **Use mtar for creating or extracting a tar file**

A modified gnu tar command, */usr/local/bin/mtar*, is Lustre stripe aware and will create tar files or extract files with appropriately sized stripe counts. Currently, the number of streps is set to the number of gigabytes of the file.

- **Keep copies of your source on the Pleiades home filesystem and/or Lou**

Be aware that files under /nobackup[p1,p2,p10-p60] are not backed up. Make sure that you have copies of your source codes, makefiles, and any other important files saved on your Pleiades home filesystem or on Lou, the NAS storage system.

- **Avoid accessing executables on Lustre filesystems**

There have been a few incidents on Pleiades where users' jobs encountered problems while accessing their executables on /nobackup. The main issue is that the Lustre clients can become unmounted temporarily when there is a very high load on the Lustre filesystem. This can cause a bus error when a job tries to bring the next set of instructions from the inaccessible executable into memory.

Executables run slower when run from the Lustre filesystem. It is best to run executables from your home filesystem on Pleiades. On rare occasions, running executables from the Lustre filesystem can cause executables to be corrupted. Avoid copying new executable over existing executables of the same within the Lustre filesystem. The copy causes a window of time (about 20 minutes) where the executable will not function. Instead, the executable should be accessed from your home filesystem during runtime.

- **Increase the stripe_count for parallel writes to the same file**

When multiple processes are writing blocks of data to the same file in parallel, I/O performance is better for large files when the stripe_count is set to a larger value. The stripe count sets the number of OSTs the file will be written to. By default, the stripe count is set to 1. While this default setting provides for efficient access of metadata for example to support "ls -l"—large files should use stripe counts of greater than 1. This will increase the aggregate I/O bandwidth by using multiple OSTs in parallel instead of just one. A rule of thumb is to use a stripe count approximately equal to the number of gigabytes in the file.

It is also better to make the stripe count be an integral factor of the number of processes performing the write in parallel so that one achieves load balance among the OSTs. For example, set the stripe count to 16 instead of 15 when you have 64 processes performing the writes.

- **Limit the number of processes performing parallel I/O**

Given that the numbers of OSSes and OSTs on Pleiades are about a hundred or fewer, there will be contention if a huge number of processes of an application are involved in parallel I/O. Instead of allowing all processes to do the I/O, choose just a few processes to do the work. For writes, these few processes should collect the

data from other processes before the writes. For reads, these few processes should read the data and then broadcast the data to others.

- **Stripe align I/O requests to minimize contention**

Stripe aligning means that the processes access files at offsets that correspond to stripe boundaries. This helps to minimize the number of OSTs a process must communicate for each I/O request. It also helps to decrease the probability that multiple processes accessing the same file communicate with the same OST at the same time.

One way to stripe-align a file is to make the stripe size the same as the amount of data in the write operations of the program.

- **Avoid repetitive *stat* operations**

Some users have implemented logic in their scripts to test for the existence of certain files. Such tests generate *stat* requests to the Lustre server. When the testing becomes excessive, it creates a significant load on the filesystem. A workaround is to slow down the testing by adding *sleep* in the logic. For example, the following user script tests the existence of the files WAIT and STOP to decide what to do next.

```
touch WAIT
rm STOP

while ( 0 <= 1 )
  if(-e WAIT) then
    mpiexec ...
    rm WAIT
  endif
  if(-e STOP) then
    exit
  endif
end
```

When neither the WAIT nor STOP file exists, the loop ends up testing for their existence as fast as possible (on the order of 5000 times per second). Adding a *sleep* inside the loop slows down the testing.

```
touch WAIT
rm STOP

while ( 0 <= 1 )
  if(-e WAIT) then
    mpiexec ...
    rm WAIT
  endif
  if(-e STOP) then
    exit
  endif
  sleep 15
end
```

end

- **Avoid multiple processes opening the same file(s) at the same time**

On Lustre filesystems, if multiple processes try to open the same file(s), some processes will not be able to find the file(s) and the job will fail.

The source code can be modified to call the sleep function between I/O operations. This will reduce the occurrence of multiple access attempts to the same file from different processes simultaneously.

```
100  open(unit,file='filename',IOSTAT=ierr)
      if (ierr.ne.0) then
        ...
        call sleep(1)
        go to 100
      endif
```

When opening a read-only file in Fortran, use ACTION='read' instead of the default ACTION='readwrite'. The former will reduce contention by not locking the file.

```
open(unit,file='filename',ACTION='READ',IOSTAT=ierr)
```

- **Avoid repetitive open/close operations**

Opening files and closing files incur overhead and repetitive open/close should be avoided.

If you intend to open the files for read only, make sure to use **ACTION='READ'** in the open statement. If possible, read the files once each and save the results, instead of reading the files repeatedly.

If you intend to write to a file many times during a run, open the file once at the beginning of the run. When all writes are done, close the file at the end of the run.

Reporting Problems

If you report performance problems with a Lustre filesystem, please be sure to include the time, hostname, PBS job number, name of the filesystem, and the path of the directory or file that you are trying to access. Your report will help us correlate issues with recorded performance data to determine the cause of efficiency problems.

Lustre Filesystem Statistics in PBS Output File

For a PBS job that reads or writes to a Lustre file system, a Lustre filesystem statistics block will appear in the PBS output file, just above the job's PBS Summary block. Information provided in the statistics can be helpful in determining the I/O pattern of the job and assist in identifying possible improvements to your jobs.

The statistics block lists the job's number of Lustre operations and the volume of Lustre I/O used for each file system. The I/O volume is listed in total, and is broken out by I/O operation size.

The following Metadata Operations statistics are listed:

- open/close of files on the Lustre file system
- stat/statfs are query operations invoked by commands such as "ls -l"
- read/write is the total volume of I/O in gigabytes

The following is an example of this listing:

```
=====
LUSTRE Filesystem Statistics
-----
      nbp10 Metadata Operations
            open      close      stat      statfs      read(GB)      write(GB)
            1057      1058      1394          0          2          14
Read    4KB    8KB    16KB    32KB    64KB    128KB    256KB    512KB    1024KB
          9      3      1      0      1      0      3      2      319
Write   4KB    8KB    16KB    32KB    64KB    128KB    256KB    512KB    1024KB
        138    13      1     11     36      9     21     37    12479
-----
Job Resource Usage Summary for 11111.pbspl1.nas.nasa.gov

      CPU Time Used      : 00:03:56
      Real Memory Used   : 2464kb
      Walltime Used      : 00:04:26
      Exit Status        : 0
```

The read and write operations are further broken down into buckets based on I/O block size. In the example above, the first bucket reveals that nine data reads occurred in blocks between 0 and 4 KB in size, three data reads occurred with block sizes between 4 KB and 8 KB, and so on. The I/O block size data may be affected by library and system operations and, therefore, could differ from expected values. That is, small reads or writes by the program might be aggregated into larger operations, and large reads or writes might be broken into smaller pieces. If there are high counts in the smaller buckets, you should investigate the I/O pattern of the program for efficiency improvements.

Tips for Improving Lustre I/O

See [Lustre Best Practices](#) for multiple tips to improve the Lustre I/O performance of your jobs.